Translating ASP to a Typed Language Without Herbrand Functions

Joachim Jansen, Gerda Janssens
firstname.lastname@cs.kuleuven.be

Dept. Computer Science, KU Leuven, Belgium

Abstract. Answer Set Programming (ASP) and FO(·) are two similar state-of-the-art languages for declarative problem solving. This paper presents an automatic transformation from ASP programs to FO(·) programs for the IDP3 system. There are two main language differences that have to be overcome. Firstly there is the introduction of types, which are mandatory in FO(·) but not present in ASP. Secondly there are Herbrand function terms that are present in ASP but not sufficiently supported in IDP3. In this paper we give a short overview of both languages and introduce our transformation that overcomes these differences in the respective languages.

1 Introduction

The domain of knowledge representation [4] has had an impressive increase in interest from several research communities. Amongst current state-of-the-art declarative solving techniques are the language of ASP [15] and the language of the IDP3 system [8], which is called FO(·). The ASP community is active in the development of improving the state-of-the-art in declarative solving. Illustrating this are the yearly returning ASP competitions [12, 6, 1]. Recent efforts to standardize the different versions of ASP languages that are supported across the different systems resulted in a standardized input language format, called ASP-Core-2.

IDP3 is an ASP-like system, but does not support ASP-Core-2 problem descriptions. Several comparisons between ASP programs and their IDP3 counterparts have been made to show that these languages show a close resemblance [3]. There are two main language differences that have to be overcome: types and Herbrand function terms. Since types are not present in ASP but are required by the IDP3 system, we discuss how we can **derive types** out of untyped ASP programs. Herbrand function terms are prevalent throughout ASP programs, but IDP3 provides insufficient support for integrating Herbrand function terms into its type system. We resolve this issue by **transforming Herbrand functions to non-Herbrand functions**, for which full support is provided by IDP3 and its type system. This paper presents an automatic transformation of ASP-Core-2 ASP programs into IDP3 programs.

In section 2 we give a short overview of both the ASP and IDP3 languages and their differences. Section 3 discusses analysis techniques and shows how we

derive types out of ASP programs. More specifically, section 3.4 discusses how we transform Herbrand function terms that would be present in these types. Section 4 shows the translations, based on all the information we derived from the ASP program. Section 5 discusses related and future work. We conclude in section 6.

2 Preliminaries

In this section we give a short overview of the source language ASP, as well as a short description of the target language (FO(·)) and system (IDP3).

2.1 Answer Set Programming

In practice, there are a bunch of ASP systems that each have their own parser. Each parser supports some version of the ASP language, and optionally some extra features that the system offers. An example of this are the `#hide.` and `#show pred/arity.` constructs supported by the grounder gringo [16]. The fourth ASP competition in 2013 [2] described a basic set of functionalities that ASP systems need to support in order to compete. Part of this description was the agreement upon a standardized input language format, called ASP-Core-2 input language. For this standardized input language format a selection of essential and common language constructs in ASP systems was made. Here we discuss the language constructs present in the latest revision of this standardization [5][1]. Below we give a short version of this language specification, leaving out some of the less important details.

Syntax We use $\odot$ to denote one of the possible relational operators $\odot \in \{<, \leq, =, \neq, >, \geq\}$. We use $\otimes$ to denote one of the possible arithmetic operators $\otimes \in \{+, -, \times, \div\}$. We use t_i for terms, l_i for literals, $x_{(i)}$ for variables, $\overline{x}_{(i)}$ for variable tuples, and $p_{(i)}$ for predicates. We use $\overline{x}$ as shorthand for the union of all variables in $\overline{x}_i$ in some context. A *term* is one of the following

- a constant c with c an integer, string, or symbolic constant
- a variable x that has a name starting with an uppercase letter
- an arithmetic term $t_1 \otimes t_2$
- a functional term $p(t_1, \ldots, t_n)$, also called a *Herbrand function term*

A *literal* is one of the following

- a possibly negated predicate atom $(\neg)p(t_1, \ldots, t_n)$ with p a predicate with arity n[2]

[1] The full ASP-Core-2 language specification on which this paper is based can be found at `www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`.

[2] ASP programs support two types of negation: negation as failure and classical negation. We currently only support classical negation. This will be further discussed in section 5.

– a possibly negated built-in atom $(\neg)t_1 \odot t_2$
– an aggregate literal $\#agg\{el_1; \ldots; el_n\} \odot t$ with $agg \in \{count, sum, max, min\}$ and each el_i an aggregate element.

An *aggregate element* is of the form $t_1, \ldots, t_m : l_1, \ldots, l_n$ with each t_i a term and each l_i a non-aggregate literal. An ASP program Π is a collection of rules. An ASP*rule* has the form $h \leftarrow l_1 \wedge \cdots \wedge l_n$. With h the rule head and $l_1 \wedge \cdots \wedge l_n$ the *body formula* consisting of a conjunction of literals. Any literal in the body conjunction can use any variable that is used in the rule head. There are four possible types of rule heads. Each type of rule head determines of what type the rule is and what semantics it follows. A *rule head* is one of the following[3]:

– for an *integrity constraint*: $\emptyset$
– for a *simple rule*: a predicate atom h,
– for a *choice rule*: a bounded set expression of the form

$$n_1 \odot_1 \{p_1(\overline{x}_1) : \phi_1[\overline{x}_1]; \ldots; p_n(\overline{x}_n) : \phi_n[\overline{x}_n]\} \odot_2 n_2. \tag{1}$$

When a simple rule has a body formula that contains no literals, we also call that rule a *fact*. We use $\psi[\overline{x}]$ to shorten body formulas with $\overline{x}$ the union of all variables that occur in the head of the rule. We use $\overline{x}^{body}$ to indicate the set of variables that occur in the body, but not in the head of the ASP rule. A variable binding θ is a mapping of variables to terms. If a variable binding θ makes a body formula $\psi[\overline{x}]$ evaluate to true, we say that the rule is **activated** by that binding. In the remainder of this text, choice rules will also be called *bounded choice rules*, to stress that they impose bounds on the generated predicates.

Semantics The computational task that is typically associated with ASP programs is that of *model expansion*: a partial assignment is completed using one of its stable models [17]. For some of the more complex ASP language constructs we give an informal semantics below. Our intention is to provide the reader with an intuition on how these constructs can be translated into our target language. For a full discussion on syntax and semantics, we refer the reader to the ASP-Core-2 report [5].

Built-in atoms The truth value of built-in relational operators is given by the traditional Prolog term ordering.

Bounded Choice Rules Choice rules are used in ASP to introduce non-determinism in the rule derivation. Generally, when a body formula is activated for some binding, the head of that rule also becomes true for that binding. This interaction is more complex for choice rules. A bounded choice of the form shown in (1) indicates that the predicates $p_i(\overline{x}_i)$ can become true for bindings for which the body of the rule (denoted $\psi[\overline{x}]$) evaluates to true, respecting the following conditions:

[3] We consider weak constraints and rules with disjunctive heads out of scope for our transformation. They will be discussed in section 5.

– for each instance of $p_i(\overline{x}_i)$ that becomes true, $\phi_i[\overline{x}_i]$ must also be true
– the number of instances that becomes true (say, j) is bounded by $n_1 \odot_1 j$ and $j \odot_2 n_2$.

Most often, it holds that $n_1 \leq n_2$ and that $\odot$ is $\leq$. In this case, the choice rule states that for every binding that activates the rule, between n_1 and n_2 head elements are generated nondeterministically.

Example 1. The following choice rule is based on the "Valves Location Problem" from the 2013 ASP competition:

$$1 \leq \{\texttt{closed\_valve(v(X,Y),broken(A,B)):symm\_pipe(X,Y)}\} \leq 1 \leftarrow \texttt{pipe(A,B).} \quad (2)$$

Which means that for every pipe from `A` to `B`, this rule derives exactly one instance of `closed_valve(v(X,Y),broken(A,B))` that chooses its binding for `X` and `Y` nondeterministically, fulfilling the requirement that there is an undirected pipe going from `X` to `Y`.

2.2 The IDP System and FO(·)

An extensive description of the IDP3 system is available at [8]. In this section we first give a quick overview of a basic IDP3 program. We then identify the biggest differences with ASP programs.

Whereas an ASP program is a set of rules, an IDP3 program consists of a number of different components: vocabulary, structure, theory, and procedures.

– A **vocabulary** $\mathcal{V}$ block contains the types T and symbols Σ used in the program. A symbol can be a predicate or a function. Each predicate p/n (we use $/n$ to indicate the predicate has arity n) is typed using `type` that maps a predicate to an n-tuple of types. Each function f/n (f has n input arguments) is typed using `type` that maps a function to an $n+1$-tuple of types.
– A **structure** over vocabulary $\{T, \Sigma\}$ gives an interpretation I for T and Σ. A type is interpreted by a set of *domain elements* that represents all valid instances of this type. We use t^I to indicate the interpretation I of type t. A well-typed instance of symbol s/n is an n-tuple $\{d_1, \ldots, d_n\}$ such that $\texttt{type}(s) = \langle T_1, \ldots, T_n \rangle$ and $d_i \in T_i$. A predicate symbol is interpreted by mapping all well-typed instances to a truth value. A truth value is one of **t** (true), **f** (false), or **u** (unknown). We represent the interpretation of predicate p/n using $p^I : \{d_1, \ldots, d_n\} \mapsto \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. A function symbol is interpreted by mapping all well-typed instances to an element of its output type T_{out}. We represent the interpretation of function f/n using $f^I : \{d_1, \ldots, d_n\} \mapsto d_{out}$. We call an interpretation two-valued if predicate symbol interpretations only map to **t** or **f** and function symbol interpretations provide a mapping for all valid instances. Otherwise, the interpretation is called three-valued. Order relation $\leq_p$ depicts a precision order on truth values : $\mathbf{u} \leq_p \mathbf{f}$ and $\mathbf{u} \leq_p \mathbf{t}$. We extend this precision order to interpretations: $I_1 \leq_p I_2$ holds if and only if I_1 and I_2 have the same type interpretations

and $p^{I_1}(\{d_1, \ldots, d_n\}) \leq_p p^{I_2}(\{d_1, \ldots, d_n\})$ for every predicate $p \in \Sigma$ and well-typed instance $\{d_1, \ldots, d_n\}$ of p.

- A **theory** over vocabulary $\{T, \Sigma\}$ contains the set of constraints C and the definition Δ which must be satisfied. [4] C contains FO($\cdot$)*formulas*. An FO($\cdot$) formula is any arbitrary first-order formula where variables are assigned a type $t \in T$ and any term is allowed to be an aggregate term. Definition Δ contains a set of rules R or the form

$$\forall \overline{x} : p(\overline{x}) \leftarrow \phi[\overline{x}].$$

  Where $p \in \Sigma$ is a predicate and ϕ a FO($\cdot$) formula with free variables $\overline{x}$. A theory $\mathcal{T}$ is satisfied in a structure I, denoted $I \models \mathcal{T}$ if every constraint $c \in C$ evaluates to $\mathbf{t}$ under I and if I interprets every defined predicate in Δ by its Well-Founded Model.
- A **procedure** contains any Lua [18] script code, augmented with the built-in subroutines that IDP3 offers. For the purpose of this paper it is sufficient to only consider the *model expansion* subroutine. The result of a model expansion call `sol = modelexpand(`$\mathcal{T}$`,` $\mathcal{I}_{in}$`)` is a two-valued structure more precise than $\mathcal{I}_{in}$ in which $\mathcal{T}$ is satisfied. In short, the following properties hold for `sol`:
  - $\mathcal{I}_{in} \leq_p$ `sol`
  - `sol` is two-valued
  - `sol` $\models \mathcal{T}$

After this short overview of the IDP3 system, we list the main differences with ASP systems we encountered during the translation and name them. More specifically we discuss the required parts of an IDP3 specification not present in ASP programs:

A1 FO($\cdot$) is typed and, before calling modelexpand, the input structure must provide an interpretation for all types in the specification
A2 FO($\cdot$) formulas can be any arbitrary first-order logic formula

as well as required ASP functionality that is not offered by the IDP3 system:

A3 term ordering
A4 Herbrand function terms

What follows is a description of translation that was implemented to translate ASP programs into IDP3 programs.

3 Type Derivation

As we discussed in the introduction, most ASP programs will have implicit typing. It is difficult, perhaps impossible, to automatically extract exactly those

[4] We assume there is only one definition. Any theory containing multiple definitions can be rewritten to contain only one definition [24].

types the user implicitly intended. Instead, our approach constructs an upper bound for all head predicates that can be generated using the given ASP program. We transform these upper bounds to the interpretation of the resulting IDP3 types.

State-of-the-art ASP grounders [16] use an approximative method to impose an upper bound on which heads can be generated. This method removes any negative, non-deterministic predicate occurring in the body of a rule. Doing so relaxes the constraint that determines which variable instances are activated for this rule. In order to construct our upper bound, we need this approximation as well.

Example 2. Continuing on the encoding of "Valves Location Problem", the following rule derives (in part) which pipes are reachable:

$$\begin{aligned} \texttt{reached(pipe(A,B),broken(X,Y))} \leftarrow\ & \texttt{tank(A)} \wedge \\ & \texttt{pipe(X,Y)} \wedge \\ & \texttt{pipe(A,B)} \wedge \\ & \neg\texttt{closed\_valve(v(A,B), broken(X,Y))}. \end{aligned} \tag{3}$$

The predicate `closed_valve` occurs in a negative context in this rule body. Since this predicate is generated nondeterministically in Example 1, it has to be filtered out, resulting in the following rule.

$$\texttt{reached(pipe(A,B),broken(X,Y))} \leftarrow \texttt{tank(A)} \wedge \texttt{pipe(X,Y)} \wedge \texttt{pipe(A,B)}. \tag{4}$$

The high-level workflow of our type derivation is as follows:

1. determine which predicates are non-deterministic
2. create an approximative XSB program and query it for the upper bound on generated head predicates
3. translate resulting XSB values to IDP3 domain elements
4. convert the IDP3 domain elements into IDP3 types, type interpretations, and interpretations for the deterministic predicates.

3.1 Dependency Analysis

The deterministic predicates are predicates whose value can be completely calculated for any given set of input facts. These deterministic predicates are also identified in IDP3 as *input*$*$ predicates [19] and efficiently calculated using a connection with XSB [22]. There are three reasons a predicate is considered non-deterministic: i) instances of the predicate are generated using choice rules, ii) the truth value of the predicate depends (indirectly) on the truth value of a non-deterministic predicate, or iii) the predicate depends on itself through a loop over negation.

Below we give a high-level algorithm to determine which predicates are non-deterministic. We consider a more detailed specification of this algorithm out of scope for this paper.

1. start with set Σ_{nondet} empty

2. add all predicates that can be generated by choice rules to Σ_{nondet}
3. construct the dependency graph of the ASP program
4. find all loops in the dependency graph which go through a non-zero, even number of negations, and add their startpoint to Σ_{nondet}
5. find all predicates for which the dependency graph contains a path to a predicate in Σ_{nondet} and add them Σ_{nondet}
6. return Σ_{nondet}

3.2 Construction of Upper Bound Generator

A translation from an FO($\cdot$) definition to XSB exists already [19]. We obtain the approximative XSB program by transforming the ASP program to an FO($\cdot$) definition and assume that the existing translation to XSB is performed afterwards. We consider two types of rules for our XSB program: simple rules and bounded choice rules. We do not consider integrity constraints because they do not generate a head predicate.

A simple rule of the form $h(\overline{x}_h) \leftarrow \psi[\overline{x}]$. with $\psi[\overline{x}]$ a conjunction is transformed to the FO($\cdot$) rule

$$\forall \overline{x}_h : h(\overline{x}_h) \leftarrow \exists \overline{x}^{body} : approx(\psi[\overline{x}]).$$

Where $approx(\psi[\overline{x}])$ filters out negative non-deterministic predicates as shown in Example 2. If there are no body variables ($\overline{x}^{body}$ is empty) we leave out the existential quantification in the body of the rule.

A bounded choice rule of the form

$$n_1 \odot_1 \{p_1(\overline{x}_1) : \phi_1[\overline{x}_1]; \ldots; p_n(\overline{x}_n) : \phi_n(\overline{x}_n)\} \odot_2 n_2 \leftarrow \psi[\overline{x}].$$

is transformed to the FO($\cdot$) rule

$$\forall \overline{x}_i : p_i(\overline{x}_i) \leftarrow \exists \overline{x}^{body} : approx(\psi[\overline{x}] \wedge \phi_i[\overline{x}_i]).$$

for each $1 \leq i \leq n$, with *approx* as described above.

All variables in the resulting FO($\cdot$) rules are over a special type that will never fail a typecheck (but is not able to enumerate all its valid instances)[5]. The FO($\cdot$) rules show how we introduce quantifiers (as part of resovling issue A2). The quantifiers shown here are an explicit representation based on the way variables are implicitly quantified in the original ASP rule.

All resulting FO($\cdot$) rules are joined into one definition and transformed into the approximative XSB program. For each head predicate p/n in the resulting FO($\cdot$) definition we create n additional XSB rules:

```
p1(X) :- p(X,_,...,_)
p2(X) :- p(_,X,...,_)
   ...
pn(X) :- p(_,_,...,X)
```

[5] In the implementation, we skip the transformation to FO($\cdot$) and directly translate to XSB, which is also untyped.

which allows us, once our XSB program is loaded, to calculate all possible values of the i-th type of p using the query `?-`p_i`(X)`. XSB values can be integers, strings, and Herbrand function terms[6]. For the remainder of this section, we will indicate the set of XSB values resulting from the query p_i as `ans(`p_i`)`. An example of such a set is $\{1, \texttt{nil}, 2, \texttt{pos(1,4)}, \texttt{pos(5,nil)}\}$. Whilst it seems not very intuitive that a rule would generate such a combination of values, this cannot be excluded since ASP-Core-2 places no such restriction either.

3.3 Translating XSB Values to IDP3 Domain Elements

In this section we describe the function `toDomel` that transforms XSB values to IDP3 domain elements. We use `domels(ans(`p_i`))` to denote the result of applying this transformation on the set `ans(`p_i`)`.

Integer or string values can be trivially translated to a domain element, as the IDP3 system provides support for domain elements of this type as well. IDP3 also supports domain elements of the form $f_d(d_1, \ldots, d_n)$ with f_d a constructor function and d_i domain elements. We translate Herbrand function terms by creating a unique f_d for each Herbrand function f_h in `ans(`p_i`)` and applying f_d to the transformation of the Herbrand function arguments. Note that this does not imply that f_d can be used as a Herbrand function to map tuples of domain elements $\{d_1, \ldots, d_n\}$ to $f_d(d_1, \ldots, d_n)$.

3.4 Creating the IDP3 Types, Vocabulary, and Input Structure

We initialise the vocabulary $\mathcal{V}(T, \Sigma, \texttt{type})$ and input structure I_{in} as empty and fill them in the following manner.

Type construction Because we can make no assumptions whatsoever about p/n's types, we are forced to create n **unique types**. We denote the i-th newly created type of p/n as T^{p_i}. We add each T^{p_i} to T and for each p/n we add p to Σ and insert $p \mapsto \langle T^{p_1}, \ldots, T^{p_n} \rangle$ into `type`. In the structure, we set the interpretation of the T^{p_i} to `domels(ans(`p_i`))`. For Herbrand function terms that occur in `domels(ans(`p_i`))`, we do the following:

- For each Herbrand function f_h/n that used in `ans(`p_i`)`, we create and add to T the input type T^{f_j} for $1 \leq j \leq n$ and one output type $T^{f_{out}}$
- We add f_h to Σ and insert $f_h \mapsto \langle T^{f_1}, \ldots, T^{f_n}, T^{f_{out}} \rangle$ into `type`
- for each occurrence of $f_h(v_1, \ldots, v_n)$ in `ans(`p_i`)` we add $\{d_1, \ldots, d_n\} \mapsto f_d(d_1, \ldots, d_n)$ to the interpretation of f_h in I_{in}, with $d_j =$ `toDomel(`v_j`)`.

Example 3. Given that the following XSB values are the upper bound for an argument of some predicate:

$$\{\texttt{pos(1,3)}, \texttt{pos(1,4)}\} \tag{5}$$

[6] Our communication with the XSB interface returns everything as a string, so we cannot make the distinction between symbolic constants and strings. We derive the type of an XSB value based on the string content.

We create types T^{pos_1} and T^{pos_2} to represent the type of the values at index 1, respectively 2. Type $T^{pos_{out}}$ is created to represent the valid output values. We then create and insert the function *pos* into Σ, with `type(`*pos*`)`$=\langle T^{pos_1}, T^{pos_2}, T^{pos_{out}}\rangle$. We insert the following interpretations into I_{in}:

- $(T^{pos_1})^{I_{in}} = \{1\}$
- $(T^{pos_2})^{I_{in}} = \{3, 4\}$
- $(T^{pos_{out}})^{I_{in}} = \{pos_d(1,3), pos_d(1,4)\}$
- $pos^{I_{in}} = \{(1,3) \mapsto pos_d(1,3), (1,4) \mapsto pos_d(1,4)\}$

Additionally, we can use XSB to already set the interpretation for all predicates that are not in Σ_{nondet}. We do this by querying XSB with `?- p(`$\mathtt{X}_1$`,...,`$\mathtt{X}_n$`).` and setting the interpretation of p in I_{in} such that it maps to true (**t**) for any of the returned instances and maps to false (**f**) for any other instance.

4 Translation to FO(·)

In this section we translate the ASP rules to an IDP3 theory $\mathcal{T}$. We initialise $\mathcal{T}$ to an empty set of constrains ($\mathcal{T}_C$) and one definition ($\mathcal{T}_\Delta$) with an empty set of rules. The translation is composed of several steps; on the the term level (`transTerm(`t`)`), the literal level (`transLit(`l`)`), and the rule level (`transRule(`r`)`). Each of these steps has access to the following global variables containing the information we have derived in previous steps:

- The set Σ_{nondet} of non-deterministic predicate symbols
- A function `typeOf` that maps an ASP predicate p and index i to T^{p_i}
- A function `toDomel` that maps ASP values to IDP3 domain elements[7]
- A function `toFunc` mapping Herbrand functions f_h to their IDP3 version
- Vocabulary $\mathcal{V}(T, \Sigma, \mathtt{type})$ and interpretation I_{in}
- A (trivial) map `toIDP` to translate ASP built-in relational operators $\odot$, arithmetic functions $\otimes$, and aggregate functions *agg* to their IDP3 counterparts.

4.1 Translating Terms

The rewrite rules that `transTerm(t)` performs are depicted in Figure 1. `createVar` maps ASP variables to IDP3 variables. We do not yet introduce types for the variables here - we discuss how to do this at the end of the rule translation step.

In addition to translating terms, we provided support for term ordering, as is present in ASP programs.

[7] Here we reuse the `toDomel` transformation used to handle XSB values.

$$
\begin{array}{rl}
v & \mapsto \texttt{createVar}(v) \\
c & \mapsto \texttt{toDomel}(c) \\
t_1 \otimes t_2 & \mapsto \texttt{transTerm}(t_1)\ \texttt{toIDP}(\otimes)\ \texttt{transTerm}(t_2) \\
f(t_1 \dots t_n) & \mapsto \texttt{toFunc}(f)\ (\texttt{transTerm}(t_1) \dots \texttt{transTerm}(t_n))
\end{array}
$$

Fig. 1. `transTerm`(t): transformations from ASP terms to IDP3 terms

4.2 Translating Literals

Figure 2 shows the transformations that are performed by `transLit`. If the input literal is negated, we return the negation of the resulting FO$(\cdot)$ formula. IDP3 supports *enumerated* set expressions $[s_1; \dots; s_n]$ with s_i a set expression, and *quantified* set expressions $\{\overline{x} : \phi[\overline{x}] : t[\overline{x}]\}$. The transformation of ASP aggregate literals contains an enumerated set expression, enumerating over the results of the subroutine `qset`(el) that transforms an ASP aggregate element to its IDP3 counterpart. This is done by transforming $el = t_1, \dots, t_m : l_1, \dots, l_n$ to the IDP3 quantified set expression $\{\ \texttt{fv}(el)\ :\ \bigwedge_{1 \leq i \leq n}\ l_i\ :\ t_1\ \}$ with `fv`(el) an operation that results in the free variables of el.

$$
\begin{array}{rl}
p(t_1 \dots t_n) & \mapsto p(\texttt{transTerm}(t_1) \dots \texttt{transTerm}(t_n)) \\
t_1 \odot t_2 & \mapsto \texttt{transTerm}(t_1)\ \texttt{toIDP}(\odot)\ \texttt{transTerm}(t_2) \\
agg\{el_1; \dots; el_n\} \odot t & \mapsto \texttt{toIDP}(agg)[\texttt{qset}(el_1); \dots; \texttt{qset}(el_n)]\ \texttt{toIDP}(\odot)\ \texttt{transTerm}(t)
\end{array}
$$

Fig. 2. `transLit`(l): transformations from ASP literals to IDP3 formulas

4.3 Translating rules

Algorithm 1 shows pseudocode for the `transRule` subroutine. All input that was declared at the beginning of this section, as well as the resulting sets of constraints ($\mathcal{T}_C$) and definition ($\mathcal{T}_\Delta$) are considered global variables. Line 2 shows how every rule translates its body in the same way: body literals are translated and their results are put back together as a conjunction, quantified over by an existential quantification over the body variables if necessary. Line 5 shows that simple rules that define deterministic predicates are not translated, since their interpretation was already calculated during the type derivation step. Translation is trivial for simple rules (line 6: translate head literal, construct IDP3 rule) and integrity constraints (line 8: add negated body translation as constraint), but choice rules are more difficult: `transChoiceRule` transforms bounded choice rules of the form

$$n_1 \odot_1 \{p_1(\overline{x}_1) : \phi_1[\overline{x}_1]; p_2(\overline{x}_2) : \phi_2(\overline{x}_2); \dots; p_n(\overline{x}_n) : \phi_n(\overline{x}_n)\} \odot_2 n_2\ \texttt{:-}\ \psi[\overline{x}].$$

into FO$(\cdot)$ constraints and definition:

$\{\forall \overline{x} : S(\overline{x}) \ \leftarrow \psi[\overline{x}].\}$
$\forall \overline{x} : S(\overline{x}) \quad \Rightarrow n_1 \odot_1 \{p_1(\overline{x}_1) : \phi_1[\overline{x}_1]; p_2(\overline{x}_2) : \phi_2(\overline{x}_2); \ldots; p_n(\overline{x}_n) : \phi_n(\overline{x}_n)\} \odot_2 n_2$
$\forall \overline{x}_i : p_i(\overline{x}_i) \Rightarrow \exists \overline{x} \setminus \overline{x}_i : S(\overline{x}) \wedge \phi_i(\overline{x}_i)$.

We add these constraints to $\mathcal{T}_C$ and add the rule in the definition to $\mathcal{T}_\Delta$.

```
   input: ASP Rule r
 1 Function transRule(r):
 2 |  b ← ∃bodyvars(r) : ⋀_{l_i ∈ body(r)} transLit(l_i)
 3 |  switch r do
 4 |  |  case Simple Rule
 5 |  |  |  if headsymbol(r) ∈ Σ_nondet then
 6 |  |  |  |  add transLit(head(r)) ← b to R
 7 |  |  case Integrity Constraint
 8 |  |  |  add ¬b to C
 9 |  |  case Bounded Choice Rule
10 |  |  |  transChoiceRule(r, b)
```

Algorithm 1: pseudocode for `transRule`

The new rule defines which instances of S (a newly introduced unique symbol) are "activated" by the body. The first constraint says that if a head is "activated", the correct number of specific head symbols must be made true (if-constraint). The second constraint says that if a specific head symbol is made true, it must be so that it is "activated" by its body (only-if-constraint).

Types of variables After an ASP rule has been translated using `transRule(`r`)`, we give types to all variables in the following manner. If a variable v occurs in a head predicate p of the original ASP rule on position i, we give it type T^{p_i}, derived for that head predicate in that position. If a variable only occurs in the body, in some body predicate b on position i, we give it the type T^{b_i}, the derived type for that body predicate position.[8]. We choose one T^{b_i} at random where it may be that there is only a very small subset of the instances of T^{b_i} that activate the rule. The type we give to body variables is not optimal in the sense that it could be a type containing more instances than those that will activate the rule. Since grounding in FO$(\cdot)$ is generally done by instantiating variables with all values in their types, translating to types with fewer instances is considered better because it will positively impact grounding performance.

5 Related and Future Work

Prolog [21, 20], as well as Datalog [25, 14] have a large history with respect to typing logic programs. However, this work is not immediatly applicable to the

[8] We consider problems where body variables do not occur in predicates out of scope.

type derivation performed in this paper. Some of these works assume a user-made typing of input facts, whilst others do not create subtypes for symbolic constants, which is essential for our approach.

We intend to further explore this topic in several ways. First, we want to generalize our translation so that some of the assumptions made here can be lifted. For weak constraints, we can transform rules of this kind into an IDP3 optimization statement. Regarding rules with disjunctive heads, this will prove a more difficult challenge. Because deciding whether a disjunctive logic program has a model falls into the second order of the polynomial hierarchy, higher-order language constructs and reasoning techniques need to be provided in IDP3. Support of negation as failure is also challenging. IDP3 works with the Well-Founded Semantics [23] (and with good reason [11]), whereas ASP uses Stable Semantics [17]. These two semantics differ only when it comes to loops over negation. In addition to choice rules, ASP programs use these loops over negation in combination with negation as failuse to generate nondeterministic predicates. Since IDP3 supports another way of inserting nondeterminism (the uncertainties are directly specified in the structure), IDP3 has no need of these loops over negation or negation as failuse. How to properly map ASP loops over negation and negation as failure to IDP3's way of specifying uncertainties is part of ongoing work.

Additionally, we wish to further optimize our translation result. This can be achieved by selecting a smaller type for body variables, as is discussed in section 4.3. Finally, we also intend to perform experiments that compare the performance the native ASP solvers with our translation and IDP3 configuration. Because the described transformation itself is a computationally intensive task (querying the values of types), as well as the possibilily of of the generated types being sub-optimal, it is to be expected from these experiments that using ASP solver on the native problem specifications is superior to using IDP3 on the translated specifications. However, we propose it would be interesting to use this translation to set up experiments that compare non-default state-of-the-art solving techniques in ASP and IDP3, such as lazy grounding [7, 10], their usage of Constrain Programming technologies [9], or symmetry breaking [13].

6 Conclusion

This paper provides a description of transforming ASP programs into IDP3 programs. We briefly discuss the ASP language and FO($\cdot$), the language of the IDP3 system and characterize the differences in these languages. The main differences are that ASP supports Herbrand function terms, but does not support types, whilst IDP3 demands that types are present, but does not fully support herbrand function terms.

Acknowledgements

We thank the reviewers for their insightful and constructive comments.

References

1. Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, Jörg Pührer, Christoph Redl, Francesco Ricca, Patrik Schneider, Martin Schwengerer, Lara Katharina Spendier, Johannes Peter Wallner, and Guohui Xiao. The fourth Answer Set Programming competition: Preliminary report. In Pedro Cabalar and Tran Cao Son, editors, *LPNMR*, volume 8148 of *LNCS*, pages 42–53. Springer, 2013.
2. Fourth answer set competition. `https://www.mat.unical.it/aspcomp2013/`, 2013.
3. A comparison between ASP and FO(·). `https://people.cs.kuleuven.be/~matthias.vanderhallen/IDPvsASP/`.
4. Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.
5. Francesco Calimeri. ASP-Core-2 input language format. Technical report, ASP Standardization Working Group, 2013.
6. Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third open answer set programming competition. *TPLP*, 14(1):117–135, 2014.
7. Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Answer set programming with constraints using lazy grounding. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 115–129. Springer, 2009.
8. Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312, 2014.
9. Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion in the presence of function symbols using constraint programming. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, pages 1068–1075. IEEE Computer Society, 2013.
10. Broes de Cat, Marc Denecker, Maurice Bruynooghe, and Peter J. Stuckey. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. (JAIR)*, 52:235–286, 2015.
11. Marc Denecker. The well-founded semantics is the principle of inductive definition. In Jürgen Dix, Luis Fariñas del Cerro, and Ulrich Furbach, editors, *JELIA*, volume 1489 of *LNCS*, pages 1–16. Springer, 1998.
12. Marc Denecker, Joost Vennekens, Stephen Bond, Martin Gebser, and Mirosław Truszczyński. The second answer set programming competition. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *LPNMR*, volume 5753 of *LNCS*, pages 637–654. Springer, 2009.
13. Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 49–56. IEEE Computer Society, 2012.
14. Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 300–309, 1991.
15. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

16. Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo: A new grounder for Answer Set Programming. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *LPNMR*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.
17. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
18. Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.
19. Joachim Jansen, Albert Jorissen, and Gerda Janssens. Compiling input∗ FO(·) inductive definitions into tabled Prolog rules for idp3. *TPLP*, 13(4-5):691–704, 2013.
20. T. L. Lakshman and Uday S. Reddy. Typed prolog: A semantic reconstruction of the mycroft-o'keefe type system. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 202–217, 1991.
21. Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984.
22. T. Swift and D.S. Warren. XSB: Extending the power of Prolog using tabling. *TPLP*, 12(1-2):157–187, 2012.
23. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.
24. Joost Vennekens, Maarten Mariën, Johan Wittocx, and Marc Denecker. Predicate introduction for logics with a fixpoint semantics. Part I: Logic programming. *Fundamenta Informaticae*, 79(1-2):187–208, September 2007.
25. David Zook, Emir Pasalic, and Beata Sarna-Starosta. Typed datalog. In *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages*, PADL '09, pages 168–182, Berlin, Heidelberg, 2009. Springer-Verlag.